SpartyJet User's Manual

Pierre-Antoine Delsart, Kurtis Geerlings, Joey Huston,
Brian Martin, and Christopher Vermilion

May 23, 2011

Contents

1 Overview

SpartyJet is a set of software tools for jet finding and analysis, built around the FastJet library of jet algorithms. Besides physics motivations, it has been written with three goals in mind:

- ease of use
- flexibility
- extensibility

To meet these goals, SpartyJet is modular: it consists of bricks of software put together allowing one to:

- access a wide variety of input formats
- perform any operation on this input, including jet finding, modification, and measurement
- save all results and any associated quantities in a ROOT `TTree`

SpartyJet extends basic jet finding with a large library of built-in jet tools to implement input and output cuts, jet modifications such as filtering and pile-up subtraction, and jet measurements ("moments") that can be calculated for every jet and stored in the output file. A typical SpartyJet analysis consists of a script that describes a set of jet algorithms, each with a set of jet tools that modify and measure the found jets. Section 3 walks through a simple example. In addition, a graphical interface is in development that allows the user to explore jet analyses interactively.

2 Installation

2.1 Requirements

OS Most Linux distributions, Mac OS X, Cygwin presumably possible but not tested.

Compiler Tested with gcc; built with make. (New, experimental CMake build requires CMake.)

ROOT Recent version, tested with 5.26 and 5.28, but older versions probably OK.

FastJet (included) Relies on FastJet for most jet finding and some internal features. Can link against an external copy or build the built-in version.

Python (optional) Python, 2.4 or later — not necessary but highly recommended. The Python interface uses PyROOT, so you must have built ROOT with this enabled. (We anticipate that the Python interface will migrate to SWIG instead of PyROOT with a future release.)

Fortran compiler (optional) Needed to compile the libraries allowing StdHep reading. Tested with gfortran.

2.2 Compilation

To compile:

```
# Setup ROOT such that root-config is in your path
# for example source root/bin/thisroot.sh
cd spartyjet
source setup.sh
make
```

- **Options**: SpartyJet has several building options to note:
  1. **FastJet:** SpartyJet depends on FastJet for jet finding and some internal features, and the latest version is included in the SpartyJet distribution. If you prefer to use your own installation, simply add `your-fastjet/bin` to the environmental variable `$PATH` such that `fastjet-config` can be found.
     NOTE: If you have linking problems between your version of FastJet and SpartyJet, either recompile your FastJet with the `--with-pic` option enabled before compilation, or allow SpartyJet to compile its own version. Note also that to enable all FastJet plugins, you must pass the `--enable-allcxxplugins` flag to `configure`. This is done by default for the built-in version.
  2. **StdHEP libraries:** These require a `FORTRAN` compiler and are automatically compiled if you have `gfortran`,`f77`, or `g77` in your `$PATH`. If you would like to try a different compiler, set the environmental variable `$F77` to the compiler binary.
  3. **Pythia 6/8 interface:** If you have ROOT compiled with the Pythia 6 andor Pythia 8 interfaces enabled, you can use this from within SpartyJet to generate events in Pythia and feed the output directly to SpartyJet. To enable this, set the variable `$PYTHIA6DIR` andor `$PYTHIA8DIR` If you do not have `PYTHIA` support in ROOT, you can add it by doing:
     ```
     cd $ROOTSYS
     ./configure --enable-pythia6 --enable-pythia8
     ```

```
--with-pythia6-libdir=/my/pythia6/
--with-pythia8-incdir=/my/pythia8145/include/
--with-pythia8-libdir=/my/pythia8145/lib/
make
```

- **Libraries:** This will build a set of libraries in `spartyjet/libs` that you can load from a ROOT session or Python script, or you can link to to build an executable.

libs/libExternal.so	-	FastJet and other code SpartyJet depends on
libs/libJetCore.so	-	Core infrastructure
libs/libIO.so	-	Facilities for reading and writing a variety of file formats
libs/libFastJet.so	-	Tools that rely on FastJet, including jet finding
libs/libJetTools.so	-	Other JetTools
libs/libEventShape.so	-	Thrust and other event shapes
libs/libSpartyDisplay.so	-	SpartyJet GUI
libs/libExternalTools.so	-	A set of third-party jet tools, with wrappers

2.3 Running

Working examples of how to use SpartyJet can be found in the following directories:

`spartyjet/examples_py`	:	Python scripts (recommended)
`spartyjet/examples_C`	:	ROOT scripts and compiled programs, in C++

The Python interface to SpartyJet is strongly preferred, and C++ access may be deprecated in a future release. We are also planning to switch from PyROOT to SWIG for Python access to SpartyJet libraries, with ROOT only used for output. To use the Python scripts, some environment variables needs to be set, which can be accomplished via:

```
source setup.sh
```

in the `spartyjet/` directory. This exports the relevant paths to your `LD_LIBRARY_PATH` and sets the environment variable `SPARTYJETDIR`, which allows the SpartyJet Python modules and libraries to be accessible from any directory. The relevant lines of `setup.sh` could also be copied into your shell's `rc` file, e.g. `~/.bashrc`.

2.4 CMake build

The SVN trunk now includes an experimental CMake build system. If you have cmake installed, you can build SpartyJet by creating a build directory (e.g., `spartyjet/build`) and running:

```
cmake ..
make
```

In a future release, the CMake build system should allow natively building on Windows (not Cygwin), but this will likely require Windows builds for SpartyJet's dependencies.

3 A Simple Example

The simplest way to get a feel for how SpartyJet works is to consider an example; in this section we walk through the script `spartyjet/examples_py/simpleExample.py`. This script runs the anti-k_T algorithm on the first 10 events listed in `data/J1_Clusters.dat`, makes a simple measurement on the found jets, and stores the results.

- Load the libraries that are needed for the algorithms that you are running (you need to `source setup.sh` in the main directory first to set up environment variables):

```
from  SpartyJetConfig import *
```

- Create a `JetBuilder` object to manage SpartyJet jobs. The argument sets the output message level — options are {`DEBUG, INFO, WARNING, ERROR`}. Log messages throughout the code are tagged with a message level; messages with level lower than the current output level are suppressed. Note that the `SpartyJet` namespace is loaded as `SJ`.

```
builder = SJ.JetBuilder(SJ.INFO)
```

- Create an input object of type `StdTextInput` with the filename containing the events. Many types of input are available; see Section 6.

```
input = SJ.StdTextInput('data/J1_Clusters.dat')
builder.configure_input(input)
```

- Define the jet algorithm(s) that you want to run. Most algorithms are implemented in FastJet via the `FastJetFinder` tool. The three arguments are a name for the tool, the algorithm's FastJet enumeration in `fastjet::JetAlgorithm`, and the R parameter. You can also pass your own `fastjet::JetDefinition`, including a plugin algorithm; see `FJExample.py` for examples. Note that the `fastjet` namespace is loaded as `fj`.

```
name = 'AntiKt4'
alg = fj.antikt_algorithm
R = 0.4
antikt4 = SJ.FastJet.FastJetFinder(name, alg, R)
```

- Pass algorithm to your `JetBuilder`. This sets up a `JetAnalysis` object, which holds a chain of `JetTools` to be executed. A jet finder is one example of a `JetTool`; see Section 5 for many more. `add_default_analysis()` adds some default tools before and after the jet finder passed (currently negative energy correctors, if enabled). `add_analysis()` inserts a user-provided `JetAnalysis`.

```
builder.add_default_alg(antikt4)
```

- Insert another tool in the chain, in this case making a measurement of jets' angular moments in η and ϕ. These will also be stored in the output ROOT file. `add_jetTool()` can take a second argument, the name of an algorithm to add the tool to; otherwise the tool is added to all algorithms.

  ```
  builder.add_jetTool(SJ.EtaPhiMomentTool())
  ```

- Configure (optional) simple text output for quick visual check of results.

  ```
  builder.add_text_output('data/output/text_simple.dat')
  ```

- Configure the Ntuple output by specifying the name of the tree and the `ROOT` file you want the data to be stored in. This output can be manipulated via your own `ROOT` scripts or be viewed with the SpartyJet GUI — see `guiExample.py`.

  ```
  builder.configure_output('SpartyJet_Tree','data/output/simple.root')
  ```

- Give the command to run the algorithms on the first 10 events

  ```
  builder.process_events(10)
  ```

Running the script will process the first 10 events on the file specified in the input object and produce the `.root` file specified in `configure_output`. To view the output with the GUI, run

```
examples_py/guiExample.py data/output/simple.root
```

from the main directory. This is meant as a basic introduction, and there are many more functions than listed here. See the other examples for more.

4 JetBuilder

`JetBuilder` is the job manager for SpartyJet. `JetBuilder` takes the input from an `InputMaker` (see Section 6) and passes it through a sequence of `JetTools`. A sequence of `JetTools` makes up a `JetAnalysis`. The final list of jets, and associated moments, is passed to an `NtupleMaker`, which prepares the output. This is shown schematically in Fig. 1.

4.1 Input Functions

Example: `examples_py/inputExample.py`

Input is passed to `JetBuilder` via:

```
builder.configure_input(input)
```

where input can be any class deriving from `InputMaker`. See Section 6 for examples.

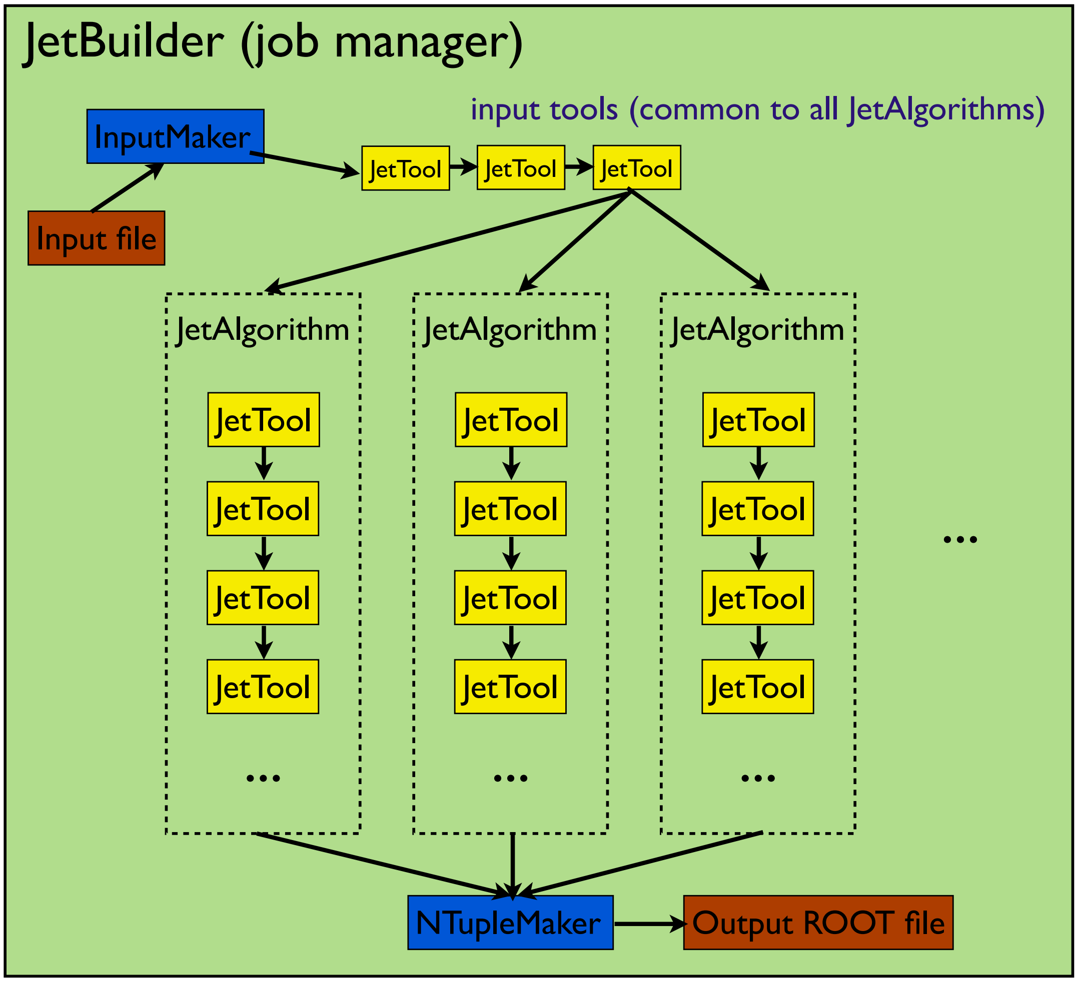

Figure 1: The structure of a SpartyJet analysis. A `JetBuilder` gets input from a file using an `InputMaker`, passes the input through a sequence of input `JetTools`, and passes the output to several `JetAnalyses` (labeled JetAlgorithms in this diagram). `JetAnalyses` consist of a sequence of `JetTools` that are run sequentially. The output of each `JetAnalysis` is passed to an `NTupleMaker` which stores the final jets and accompanying moments in a file.

4.2 Minimum Bias Overlay

Example: `examples_py/overlayExample.py`

`JetBuilder` allows the user to add minimum bias (MB) events to the signal events to study the effects of pileup. To enable MB events, one must:

1. Create another input object:
```
MBinput = SJ.StdTextInput('../data/MB_Clusters.dat')
```
2. Tell `JetBuilder` to add `n` MB events to each signal event:
```
builder.add_minbias_events(n, MBinput)
```

`JetBuilder` will start at the beginning of the MB data file and read the first `n` events for the first data event, then the second `n` events for the second data event, so on. When the end of the MB file is found, it will simply continue from the beginning. There is a third optional Boolean argument to `add_minbias_events()` that tells `JetBuilder` whether to draw the number of MB events from a Poisson distribution. In this case, `n` is the Poisson mean or expected number of MB events.

4.3 `JetTool` Options

`JetTool`s are blocks of code that act on a set of `Jets` — implemented as a `JetCollection`. A set of `JetTools` forms a `JetAnalysis`, and a `JetBuilder` holds a set of `JetAnalyses` that are each run on its inputs. Examples of specific tools can be found in the `JetTool`s section.

`JetAnalyses` are added via:
```
builder.add_default_analysis(tool)
```
where `tool` is a single `JetTool`, typically a jet finder, which forms the basis of the tool chain. Default tools are added before and after — currently just negative energy correctors, if this option is enabled.

Alternatively, to add a `JetAnalysis`, possibly with multiple `JetTool`s already associated with it:
```
builder.add_analysis(alg)
```
`JetTool`s are added to the execution sequence via:
```
builder.add_jetTool(tool)
```
or to the front of the sequence with:
```
builder.add_jetTool_front(tool)
```
These add the specified `JetTool`s to all `JetAnalyses`. A second, string argument may be passed to these methods specifying which `JetAnalysis` the tool should be added to.

A single sequence of `JetTool`s is run on the input particles before they are passed to each `JetAnalysis`. This sequence starts empty; to add to it use:
```
builder.add_jetTool_input(tool)
```
This is useful for initial tools like p_T and η cuts that are common to all algorithms.

4.4 Constituents

SpartyJet retains information about each jet's full recombination history. The method by which this information is stored is explained in Section 7. By default, this is enabled; to disable history saving, you must add `false` as a second argument:

```
builder.add_default_analysis(alg, False)
```

The full recombination history is stored in memory, but storage to disk has not yet been implemented. We hope to have this available in the near future.

4.5 Text File Output

The text file output option tells `JetBuilder` to produce an easily-readable text file that contains a list of all jets found from all algorithms for all events. To turn on the text file output, you must call `builder.add_text_output()`, and pass it the filename you want to create. For example:

```
builder.add_text_output('../data/output/text_output.dat')
```

5 JetTools

`JetTool`s are blocks of code that act on a `JetCollection` for every event. They generally perform one or more of the following functions:

- Add or remove jets from the list.
- Modify the jets themselves.
- Add information about each jet as a `JetMoment`.
- Add information about each event as an `EventMoment`.

Jet tools can be added either before or after the primary jet finder (e.g. k_{T} algorithm) in the `JetAnalysis`. The following is a list of `JetTools` shipped with SpartyJet. The relevant definitions can be found in `JetTools/`.

5.1 Selectors

These tools allow the user to remove some Jets from an input/output JetCollection, and are defined in `JetTools/JetSelectorTool.hh`.

`JetPtSelectorTool(double ptmin)` - Removes jets with p_T below cut.

`JetPtORESelectorTool(doule ptmin, double emin)` - Removes jets with p_T or energy below cuts.

`JetEtaCentralSelectorTool(double abs_eta)` - Removes jets outside of central η region.

`JetEtaForwardSelectorTool(double abs_eta)` - Removes jets inside of central η region.

`JetMassSelectorTool(double mass)` - Removes jets with m below cut.

`JetInputPdgIdSelectorTool(std::vector<int> pdgIds)` - Removes input "jets" with given PDG IDs. (Useful when the input jets are just single particles from a Monte Carlo where leptons, neutrinos, etc. are included.)

`JetMomentSelectorTool<T>(std::string momentName, T min, T max)` - Finds the given jet moment (calculated by another tool) and requires it be within (min, max). T can be any type supporting less-than comparison.

5.2 Jet and Event Moment tools

These tools calculate moments for each jet or event, which can be any type that ROOT knows how to store, most commonly `floats` or `ints`. The generic `JetMomentTool` and `EventMomentTool` calculate and store any user-implemented `JetMoment<T>` or `EventMoment<T>` object. See `JetTools/JetMomentTool.hh` for some specific examples, and `FJToolExample.py` for moment tools in action. The following are other moment-storing tools available in SpartyJet:

`HullMomentTool` This tool finds the convex hull enclosing each jet and saves the hull length and area as `Hull` and HullA respectively.

`EtaPhiMomentTool` This tool calculates angular second moments in η and ϕ of each jet stores them as `M2eta` and `M2phi`.

`PtDensityTool` This tool calculates each event's p_T density using FastJet. It does this by finding all the jets in the event with no minimum p_T requirement. It then extracts the p_T density from these jets by selecting the mean p_T density for each bin in η. These p_T densities are stored as `ptDensity` and the η bin limits are stored as `ptDensityBins`.

`JetAreaCorrectionTool` This tool uses the area of each jet and the pTDensity found by the `PtDensityTool` to calculate a correction (stored as jet moment `JetAreaCorr`) to the jet's p_T.

`YSplitterTool` This tool uses FastJet to calculate the y values associated with a set of recombinations. In this implementation, SpartyJet will run a FastJet algorithm of the user's choice on the constituents of a given jet. It can be called with either of the following constructors:

```
YSplitterTool(float R, fastjet::JetAlgorithm alg, int ny, int njet)
YSplitterTool(fastjet::JetDefinition *jet_def, int ny, int njet)
```

where `njet` is the number of jets for which the y values will be calculated and `ny` is the number of y values to calculate for each jet.

5.3 Jet Substructure Tools

Several tools to modify jet substructure have recently been proposed, and many have now been incorporated into SpartyJet. Some are implemented natively; some are essentially wrappers for existing FastJet tools. External tools that are not part of FastJet are included in the `external/` directory. Currently all of these external tools are available on the FastJet website. The file `external/README` notes which files are external and where they came from.

These tools are in active development, should be considered "beta", and are subject to change in syntax and behavior. User feedback on which features are most useful, which tools are missing, etc. is strongly encouraged. Examples of how to use these tools can be found in `examples_py/FJTool.py`.

`TopDownPruneTool` This is a generic tool implementing the "pruning" away of asymmetric branchings that is the first step in several recent substructure methods. Derived classes implement the `branch_test()` method to test if a branching represents a symmetric branching (found two subjets), an unresolved branching (e.g., the two subjets are too close together — found one subjet), or an asymmetric branching, where one subjet is discarded and the procedure recurses on the other subjet. The user can specify how many splittings to look for — one splitting will yield up to two subjets, two splittings yields up to four, etc. Two specific examples are implemented: `JHPruneTool` and `MassDropTool`. The former reproduces the first step in the Johns Hopkins top-tagging method (arXiv:0806.0848); the latter reproduces the mass-drop step in the the Higgs analysis of arXiv:0802.2470. See `JetTools/TopDownPruneTool.hh`. The constructors are:

```
JHPruneTool(double delta_p = 0.1, double delta_r = 0.19, int max_split
     = 2, string name = "JHPrune")
MassDropTool(double mu_cut = 0.67, double ycut = 0.09, int max_split =
     1, string name = "MassDrop")
```

`max_split` is the number of recursions to allow looking for symmetric splittings. `max_split = -1` continues until all branchings are unresolved. Note that this implementation of the Johns Hopkins pruning step is slightly different than the original, in that asymmetric branchings below the main splitting are discarded even if no subsequent symmetric splitting is found.

`SubjetCutTool` The various implementations of `TopDownPruneTool` all store a jet moment for the number of subjets found. This simple selector tool cuts on the number of subjets found by a given tool, and optionally unclusters the jet such that the last merging is N $\to$ 1 if N subjets are required. Construct with:

```
SubjetCutTool(TopDownPruneTool *subjet_tool, int Nsubjets = 2, bool
     uncluster = false, string name = "SubjetCut")
```

`SubjetMergeTool` If something like `SubjetCutTool` has been used to identify more than two subjets of the final jet, this tool merges the two whose combined mass is closest to a given value (m_W, say, for a top jet). If the final merging is $N \to 1$, this inserts a $2 \to 1$ merging followed by $(N-1) \to 1$. A variant of this tool is MinMassTool, which merges the two subjets with minimum combined mass and therefore does not shape the background distribution as strongly. The constructors are:

```
SubjetMergeTool(double m, string name = "SubjetMergeTool")
MinMassTool(string name = "MinMassTool")
```

FilterTool and BDRSFilterTool These wrap the Filter and FilteredJet classes written by Gavin Salam to implement jet "filtering", as in the final step of the Higgs analysis in arXiv:0802.2470. Both assume that their input JetCollection holds jets found with Cambridge/Aachen. Filtering unclusters these jets down to some angular scale Rfilt and keeps the hardest nfilt as well as any above a p_T cut ptkeep. Each jet in the input is filtered and copied into the output JetCollection. For FilterTool the Filter is created once and used for all jets; for BDRSFilterTool the filter is recreated for each jet, using $R_{\rm filt} = \min(R_{\rm filt}, \Delta R_{12}/2)$. The constructors are:

```
FilterTool(fastjet::Filter* filt, double R = 1.0, string name = "
    FilterTool")
BDRSFilterTool(double R = 1.0, double Rfilt = 0.3, int nfilt = 3,
    double ptkeep = numeric_limits<double>::max(), double rho = 0.,
    string name = "BDRSFilter")
```

If rho is non-zero, area-based subtraction is also performed. The R parameter in the constructors should be the R parameter the jets were found with; the algorithm is not re-run. See external/Filter.hh.

TopTaggerTool This tool uses a provided top-tagging class to select jets and identify their subjets. In the current implementation, TopTaggerTool is templated with a top-tagger class, and will work with either JHTopTagger or CMTopTagger, both external tools. JHTopTagger implements the Johns Hopkins top-tagging method of arXiv:0806.0848; CMTopTagger is an alternative (unpublished) method by Gavin Salam. TopTaggerTool uses the common functions maybe_top() (determines whether the jet is "tagged"), W_subjet(), and subjets(). The stored jet is taken to be the sum of subjets(), with the two subjets: the jet returned by W_subjet() and the full jet minus the W. No further substructure is stored (this is not yet available in CMTopTagger). The value of the function cos_theta_h() is stored as the jet moment "cosThetaH". The constructor is:

```
TopTaggerTool(Tagger* tagger, string name = "TopTagger")
```

Note that in Python you must supply the template argument:

```
JHtool = SJ.FastJet.TopTaggerTool(fj.JHTopTagger)(JHtagger)
```

The constructors for the top taggers are:

```
JHTopTagger(double delta_p = 0.10, double delta_r = 0.19, double mW =
    81.0)
CMTopTagger(double zmin = 0.0, double mass_max = 0.0)
```

Note that the implementation of these taggers has been changed slightly for inclusion in SpartyJet: they are set up to be constructed once and run multiple times, so the original constructors have been split into new constructors taking parameters, and a `run(ClusterSequence*, PseudoJet&)` step that runs the tagger on a given jet.

`WTaggerTool` This tool wraps the W-tagging method of Cui, Han, and Schwartz arXiv:1012.2077, which includes a large number of substructure and mass cuts. The cuts are taken from data files in `external/wtag-1.00/data`. So far there is no way to re-train the cuts from within SpartyJet— the idea is to take a pre-trained W-tagging method and plug it into a SpartyJet analysis. Since the tagger is taken "out-of-the-box", there are no input parameters:

```
WTaggerTool()
```

5.4 Miscellaneous Tools

`ForkToolParent` and `ForkToolChild` This pair of tools allows the forking of `JetTool` chains. `ForkToolParent` merely saves a copy of its input. A `ForkToolChild` is associated with a specific parent, and it reads in the `JetCollection` saved by its parent. This allows, for example, one jet algorithm to be run combined with several different jet-modifying tool chains for comparison. See `FJToolExample.py` for a usage example.

`CalorimeterSimTool` This tool applies a very simple calorimeter simulation to its input jets. Inputs are sorted into calorimeter cells on a specified η–ϕ grid. For each non-empty cell, a massless output particle is created with the direction of the cell and the total energy of all particles in the cell.

`RadialSmearingTool` This tool wraps Peter Loch's `DetectorModel` code for calorimeter simulation. For more information, see Peter's website. A usage example is given in `Boost2011.py`.

`JetNegEnergyTool` This tool is meant to be run twice: once before jet finding and once afterward. On first run, the tool finds and stores all input particles with negative energy. For each such particle it inverts the energy to be positive. On second run, the `JetNegEnergyTool` loops over jets, and for each constituent that initially had a negative energy, it corrects the jet energy by subtracting twice the constituent's (positive) energy. The `JetBuilder` method `do_correct_neg_energy(true)` inserts this pair of tools before and after all jet finders added with `add_default_analysis(tool)`; by default this is not done.

`EConversionTool` This tool simply converts the units of all the jets between MeV and GeV. The user can convert to arbitrary units as well.

6 Input

All SpartyJet jobs need an `InputMaker` object to read input from some data file and prepare a list of 4-vectors for `JetTools` to process. There are several types of `InputMaker`s available. An example of the implementation for each type of input can be seen in `examples_py/inputExample.py`. For Python scripts, the utility module `python/SpartyJetConfig.py` defines the helper function `getInputMaker(fileName)` which will create the appropriate `InputMaker` by looking at the filename extension.

6.1 NtupleInputMaker

Sample: `data/J2_clusters.root`

This form of input reads ROOT files. This `InputMaker` requires the components of the input 4-vectors to be stored in separate branches of a ROOT `TTree`. The following definitions are supported:

- `px, py, pz, E`
- `(psuedo)rapidity, phi, pt, E`
- `(psuedo)rapidity, phi, pt, m`

You need only specify how the information is stored (array or vector / float or double) and the names of the branches.

As an example, to set up an `NtupleInputMaker` to read the file `data/J2_clusters.root`, if you don't know how variables are internally stored in your ntuple, do the following:

```
root -l data/J2_clusters.root
root [0] clusterTree->MakeClass("test");
```

Open the file test.h and check to see how the variables are stored. In this example, we see lines like:

```
vector<float> *Cluster_eta;
```

indicating that our 4-vectors are stored in vectors of floats. Now to configure SpartyJet to accept this, we need to:

- Create an `NtupleInputMaker` of the correct type: (in our case: `vector<float>` for (`eta, phi, pt, E`)).

  ```
  input = SJ.NtupleInputMaker(SJ.NtupleInputMaker.EtaPhiPtE_vector_float)
  ```

  For full list of Input codes see: `JetCore/InputMaker_Ntuple.hh`

- Configure the names of the `TBranches`

  ```
  input.set_prefix('Cluster_')
  input.set_n_name('N')
  input.set_variables('eta','phi','p_T','e')
  input.setFileTree('../data/J2_clusters.root', 'clusterTree')
  ```

- Specify if input is massless, only useable in `eta,phi,pt,E` mode (if true, `pt` is ignored):

  ```
  input.set_masslessMode(True)
  ```

- Set the input file and tree names:

  ```
  input.setFileTree('../data/J2_clusters.root', 'clusterTree')
  ```

Python Shortcut:

To allow SPARTYJET to configure your Ntuple using some assumptions use the following helper function:

```
input = createNtupleInputMaker('../data/J2_clusters.root', inputprefix='
    Cluster')
```

DelphesInput

`DelphesInputMaker` is a minor extension of `NTupleInputMaker` that reads the ROOT files produced by the detector simulator DELPHES. Only calorimeter cells are read in.

6.2 StdTextInput

Sample: `data/J1_Clusters.dat`

This form of input reads ASCII files. To separate events, put one of the following lines between the events:

```
.Event
.event
N
n
```

(only the .E or .e is important in the first two).

The form of the four vectors should be:

```
E px py pz
```

This input is configured simply with:

```
input = SJ.StdTextInput('../data/J1_Clusters.dat')
```

If the form is the opposite (px py pz E), then call the function

```
input.invert_input_order(True)
```

and it will be read in properly.

An example of this input can be seen in `data/J1_Clusters.dat`

6.3 StdHepInput

Sample: `data/ttbar_smallrun_pythia_events.hep`

This form of input reads StdHEP format XDR files. It will look for particles with the status code of 1 (final state). It is also able extract the PDG ID code for each particle from the input data to allow further filtering and matching. Access to intermediate particles such as the participants in the hard scattering, or B hadrons from b quark decays, should be possible in the near future.

This input is configured simply with:

```
SJ.StdHepInput('../data/ttbar_smallrun_pythia_events.hep')
```

NOTE: To read StdHep files, you must enable StdHep compilation as explained in Section 2.2.

6.4 CalchepPartonTextInput

Sample: `data/gg_ggg_events.dat`

This form of input reads output from CalcHEP. It reads in the number of initial and final state particles, and then for each event saves only the information for the final state particles.

This input is configured simply with:

```
SJ.CalchepPartonTextInput('../data/gg_ggg_events.dat')
```

6.5 HepMCInput

Sample: `data/HepMC_sample.dat`

This form of input reads HepMC (version 2) format ASCII files. HepMC ASCII files contain the following separators:

E - Denotes new event

V - Information about a vertex

P - Information about a particle

This class reads in the 4-vectors of the particles denoted with a status code of 1 (not decayed, final state).

6.6 PythiaInput

This form of input generates and reads events directly from Pythia, without ever having to write them to a file. This requires ROOT's Pythia interface; versions 6 and 8 will both work. See `examples_py/pythiaExample.py` for an example of using Pythia in this way.

6.7 FourVecInput

A `FourVecInput` takes four-vectors from some other code; the input class is templated on a "Reader" class that must provide a simple interface for retrieving four-vectors. See `IO/FourVecInput.hh` for a complete specification, and examples_C/FourVecExample.cc for an example. This does not currently work in Python, since you would have to generate a dictionary for FourVecInput<YourReaderClass>.

6.8 Input Options

Multiple input files

The MultiInput class can be used to string a set of input files together. See examples_py/mergedInputExample.py for an example. Note that the current implementation opens all input files before beginning, which may be inefficient.

Rejecting bad input

The InputMaker can be set to remove 4-vectors with negative energy and non-physical momenta by using the following function.

```
input.reject_bad_input(bool)
```

The current default is false; no checks will be done. (An alternative to this method of dealing with bad input is to use the JetNegEnergyTool, described in the JetTools section.)

Reading of PDG ID codes

The InputMaker can be set to read PDG ID codes from the input data with the following function. This is done by default.

```
input.readPdgId(bool)
```

This makes the PDG IDs available for input selection and saves the IDs of the input particles for offline analysis.

7 Output

When using the JetBuilder class the result is a ROOT Ntuple containing jet variables for each algorithm added, plus variables for input particles to the jet algorithms. Jet and event moments are also stored in a similar way.

7.1 Output variable type

It is possible to choose the type of the variable saved in the Ntuple built by JetBuilder. Choice are between pure C array or STL vector of floats or doubles.

```
builder.output_var_style.array_type = 'vector' # (default) other option is "
    array"
builder.output_var_style.base_type = 'float' # (default) other option is "
    double"
```

7.2 Constituents

For all algorithms, constituent information can be saved as follows. Assuming an algorithm named `MyJet` has been added, two additional variables are stored in the ROOT `TTree`:

`MyJet_numC`

`MyJet_ind`

`MyJet_numC` is an array of size `MyJet_N`. `MyJet_numC[i]` is the number of constituents of `i` th jet. `MyJet_ind` is an array of size `InputJet_N`. `MyJet_ind[i]` is the index of the jet to which the `i` th input constituent has been assigned.

For example:

```
myTree.Draw('InputJet_e', 'AntiKt10_ind==0')
```

will give the energy distribution of constituents in jet number 0 (i.e. highest p_T jets) for the `AntiKt10` collection.

8 Analysis

A graphical user interface is under development for interactive SpartyJet analysis. Users can test this out with the `guiExample.py` in the `examples_py` directory. A screenshot of the GUI in action is shown in Fig. 2. The user can select which algorithms to view by ticking the boxes under "JetCollections". On the left are several event-by-event views, which will be drawn separately for each algorithm. The number of rows and columns are set in the upper left. Additional algorithms can be run (event-by-event) on the fly using the menu in the lower left.

On the right, full-run plots can be selected, which are plotted for all algorithms together (an example of this output is shown in Fig. 3). Check boxes are provided for the standard four-momentum variables, but any stored jet moment can also be plotted by entering the name in the box below, e.g. `$$_subjetM` to plot the subjet mass if the `HeavierSubjetMass` moment has been used. Note that `$$` is a placeholder for the algorithm name. Cuts can be added using `TCut` syntax, e.g. `AntiKt10_mass > 150 && AntiK10_mass < 200`. Finally, legend labels for each algorithm can be given; the syntax is ROOT `TLatex` text, e.g. `#phi_{0}` produces ϕ_0.

Contact information

Any questions/comments/suggestions, email:

Pierre-Antoine Delsart:	delsart@in2p3.fr
Joey Huston:	huston@pa.msu.edu
Brian Martin:	marti347@msu.edu
Chris Vermilion:	ckvermilion@lbl.gov

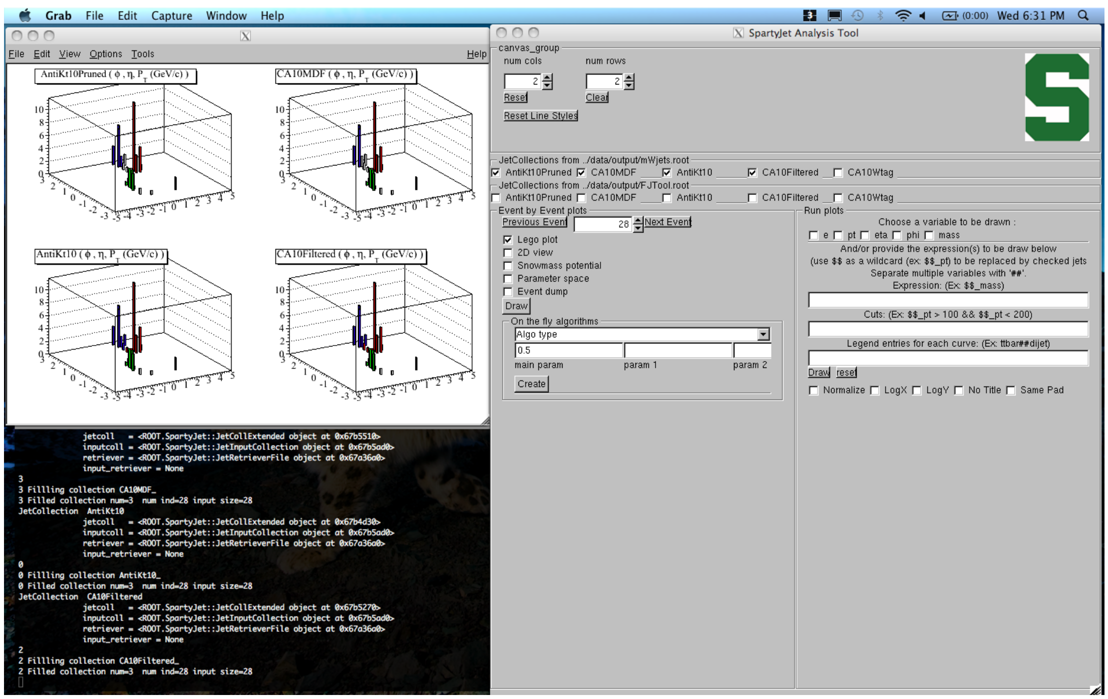

Figure 2: A screenshot of the SpartyJet graphical interface.

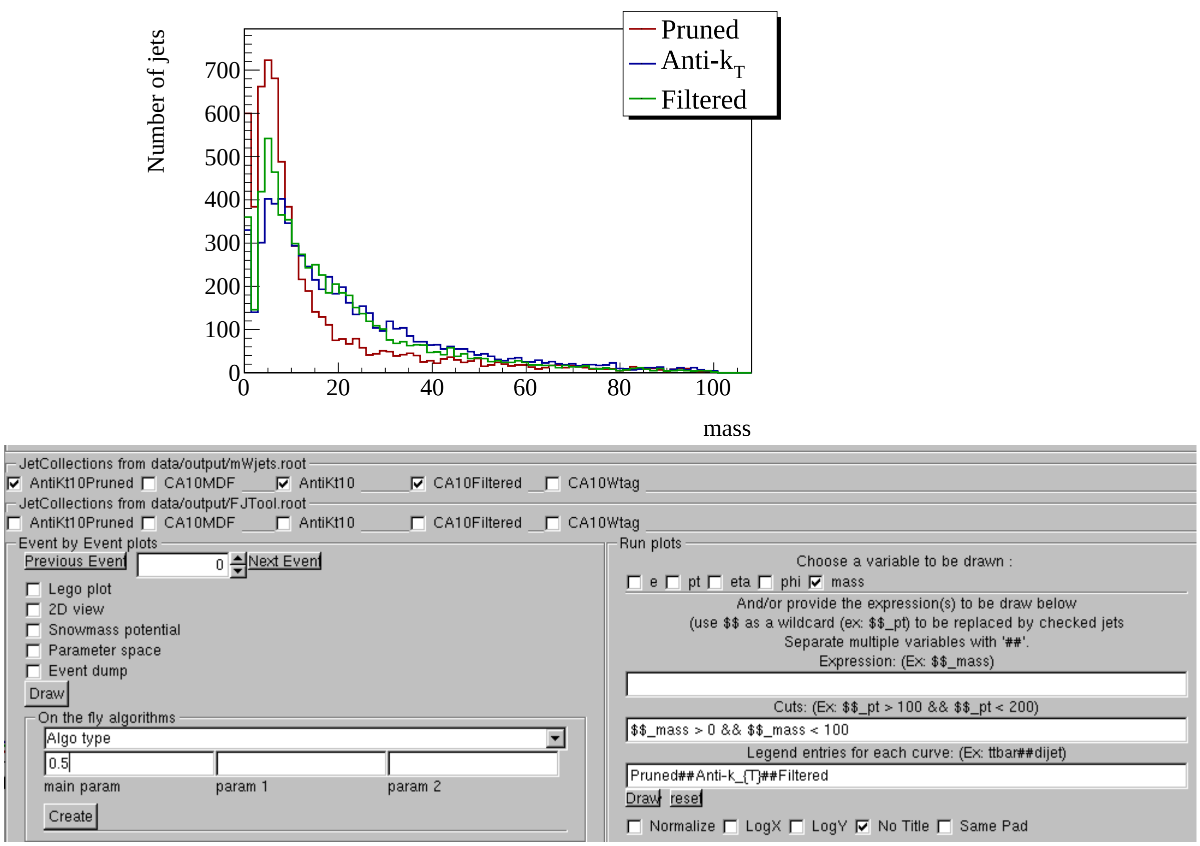

Figure 3: An example canvas produced by the run plot option (above), with the options that produced it (below).